McGill University
School of Computer Science
Sable Research Group

Using hardware data to detect repetitive program behavior

Sable Technical Report No. 2007-2

Dayong Gu and Clark Verbrugge
{dgu1, clump}@cs.mcgill.ca

March 14, 2007

Abstract

Detecting repetitive "phases" in program execution is helpful for program understanding, runtime optimization, and for reducing simulation/profiling workload. The nature of the phases that may be found, however, depend on the kinds of programs, as well how programs interact with the underlying hardware. We present a technique to detect long term and variable length repetitive program behaviour by monitoring microarchitecture-level hardware events. Our approach results in high quality repetitive phase detection; we propose quantitative methods of evaluation, and show that our design accurately calculates phases with a 92% "confidence" level. We further validate our design through an implementation and analysis of phase-driven, runtime profiling, showing a reduction of about 50% of the profiling workload while still preserving 94% accuracy in the profiling results. Our work confirms that it is practical to detect high-level phases from lightweight hardware monitoring, and to use such information to improve runtime performance.

1 Introduction

Most programs are highly repetitive; a large portion of execution time is typically spent in just one or more small code segments. Detecting, or even predicting repetitive, "phase-like" behaviour can be important for many reasons, including program understanding, identification of execution "hot spots," runtime adaptation, and so forth. Phases of course can have different properties; most phase analysis techniques concentrate on finding short-term, fixed-length phases representing periods of stable program execution. This is appropriate and reasonable for many programs, especially "regular" and scientific computations, but not necessarily appropriate for programs with more variable behaviour and/or more long-term phase structure.

Understanding performance, including the nature of program phases, of course requires understanding the underlying execution system as well as the program code. Modern processors are complicated, with many internal components and designs; pipelines, multiple-level caches, TLBs, branch predictors, multiple cores, etc. These features are very effective, but introduce a significant amount of complexity when trying to determine why a program behaves the way it does. Previous work [13, 19] has shown that there exists a tight, and often unintuitive relation between the hardware performance and the program behaviour. Hardware performance data is thus critical for developing a good understanding of program performance. Recently, and following the general maturation of hardware performance monitoring techniques in commercial machine designs, hardware event data has begun to receive more and more attention as a basis for understanding program behaviour [30], detecting program phases [4, 10, 11], and for employing adaptive optimizations [8, 16, 22].

In this paper we present a technique to detect repetitive behaviour in program execution using hardware data. Our work considers the important problem of finding long term phases of variable length, something we show is usefully present in many programs. Our design is based on creating "patterns" representing the variation in hardware event data collected from low-level hardware profilers. These patterns can then be used to detect higher-level phase changes, and incorporated into sophisticated table-based techniques to help predict program behaviour and guide runtime adaptation.

To validate our results we develop suitable metrics, as well as a sample application. We propose *Confidence* and *Possible Miss Rate (PMR)* measures to quantitatively evaluate the quality of phase detection. These calculations give a basic understanding of the quality of phase data, and are the first such measures to be formally described. We then use our phase detection results to control the runtime profiling mechanism in a Java Virtual Machine. Experimental data shows that our phase analysis is very accurate in objective measures as well as potential usage: our online phase prediction allows us to reduce about half of the

profiling workload with almost no degradation in profiling accuracy.

Specific contributions of this work include:

- We present a new phase detection technique for long-term, variable-length phases. Our design is based on hardware event data, combining a very lightweight phase analysis mechanism with an essential understanding of how current performance is strongly tied to complex hardware features.
- We describe a novel set of metrics to evaluate the quality of repetitive behaviour detection. Quantitative evaluation is important, and our metrics are the first proposed specifically for measuring the success of long-term, repetitive program phase analysis.
- As an example application of our design we demonstrate how the cost of more traditional runtime program profiling can be reduced by about half with appropriate phase information. Our optimized profiling strategy is itself a concrete runtime optimization.

The remainder of this paper is organized as follows. In the next section we describe related work on phase analysis, use of hardware data in program analysis, runtime profiling techniques and table based prediction. This is followed in Section 3 with a description of our basic phase analysis design. Our evaluation metrics are explained in detail in Section 4, with results and other experimental data reported in Section 5. We conclude and provide future work in Section 6.

2 Related Work

Most program phase detection techniques are based on comparing the differences in behaviour between fixed-length intervals; program execution is divided into short intervals and profiling data is measured in each. If the differences between two consecutive intervals is larger than a predefined *threshold* a phase transition point is declared. Hind *et al.* give a classification in [14] for this type of technique. Of course there is considerable variance in the kind of data gathered and the way data is represented. Sherwood *et al.,* for instance, make use of *Basic Block Vectors* (BBVs) to detect phase changes [24]. Following a more low level perspective, Dhodapkar and Smith use the *Instruction Working Set* to detect phase transitions [9]. Other measurement targets includes *conditional branch counts* [4], *data reuse distance* [23] and *software trace generation rate* [3].

More recently the importance of detecting phases with variable length has been considered. In [18] Lau *et al.* point out that fixed length solutions can become "out-of-sync" with the intrinsic period of the program. They show that variable length intervals are necessary in some situations [18]. The authors have also previously given a motivation for detecting periodic, long-range repetitive phases [12], and classify phase detection techniques based on the data source, from *purely software data* to *actual hardware data*. Our technique here is an instance of using actual hardware data to detect coarse, long-range repetitive phases.

Detection techniques work in a *reactive* manner; program behaviour changes are observed only after the occurrence of the change. A *predictive* mechanism is clearly more desirable for optimization purposes. Prediction techniques can be roughly divided into two types: *statistical prediction* and *table-based prediction*.

Statistical predictors estimate future behaviour based on recent historical behaviour [10]. Many statistical predictors have been developed, including (among many others) *Last value*, *Average(N)*, *MostFrequent(N)* and the *Exponentially weighted moving average predictor* (EWMA(N)). *Table-based predictors* allow for more arbitrary correlation of program activity and predicted behaviour. Mappings between events or states and predictions of the future are stored in a table and dynamically updated when large behaviour changes are

identified. Pickett and Verbrugge, for instance, develop a *memoization* predictor for *return value prediction* by correlating function arguments with likely return values [21]. Sherwood and Sair use a table-based technique to to do *Run length encoding phase prediction* based on patterns in low level branch data [25]. E. Duesterwald *et al.* give a general study on predicting program behaviour [10], comparing statistical and table-based models operating on fixed size intervals. Their experimental results show that table-based predictors can cope with program behaviour variability better than statistical predictors. Our prediction technique is largely table-based as well; we use a mixed global/local history and give prediction results with a confidence probability.

To predict events a lightweight profiling mechanism is crucial. Profiles can be obtained from program instrumentation or from a sampling scheme. *Dynamo*, for example, uses instrumentation to guide code transformations [3]. Commercial JVMs provide a basic instrumentation interface through Sun's JVMTI specification, which also makes use of instrumentation [29]. In a sampling approach, only a representative subset of the execution events are considered. Many systems use a timer-based approach to determine sampling points. On some other systems, such as IBM Tokyo JIT compiler [28] and Intel's ORP JVM [7], a count-down scheme is used, triggering sampling after n method (or other code segment) executions. Arnold and Grove present an approach that combines the timer-based and count-down schemes [2].

For our phase analysis we use the hardware counters in modern processors to gather execution data. Hardware counters provide a lightweight mechanism for gathering micro-architectural performance information that is difficult or impossible to derive from software techniques alone. This low level information can be used for guiding higher level adaptive behaviour. Barnes *et al.,* for instance, use hardware profiling to detect hot code regions and apply code optimizations efficiently [5]. Kistler and Franz describe the *Oberon* system that performs continuous program optimization [17]. They mention the benefits of using hardware counters in addition to software based techniques as crucial components of their profiling system. Other works based on hardware event information can be found in [13, 22, 30]. Hardware counters can be accessed through many software libraries, such as PMAPI [15] and PAPI [6].

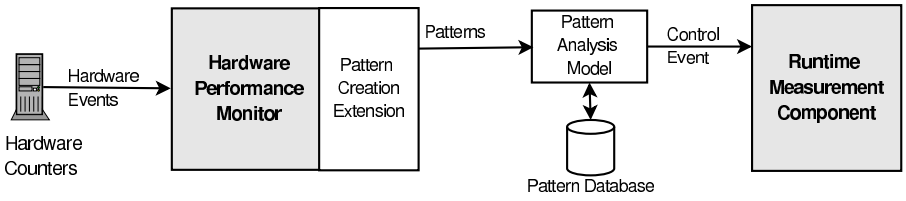

Figure 1: System structure for repetitive behaviour detection.

3 System Design

Our work is an extension to Jikes RVM [1], and Figure 1 shows an overview of the design. Raw hardware event data is read from hardware counters through the *Hardware performance monitor* (HPM), a pre-existing component in Jikes RVM. We augment the HPM with a *pattern creation extension* that generates *patterns* representing the hardware performance. This analyzes the hardware data between two consecutive sample points, summarizing the "shape" or pattern of variation in low-level performance. If we observe that the same sequence of variation in events has been encountered before, a (new) repetitive sequence will be considered. Created patterns are transferred to a *pattern analysis model* for deeper analysis. The pattern analysis model maintains a *pattern database* to store the patterns received. The pattern analysis model makes the

ultimate decision on the identification of and response to phase changes. To support our example optimization our pattern analysis model then communicates with the *runtime measurement component* in Jikes RVM, used to control the selective profiling mechanism that regulates adaptive (re)compilation. Below we describe the three main mechanisms in more detail: the *Pattern Creation Mechanism,* the *Pattern Analysis and Prediction,* and the *Profiling Control Mechanism.*

3.1 Pattern creation

There are a wide variety of properties of hardware events that could be used to detect repetitive behaviour: increasing or decreasing trends, range of variation, and distance and similarity measures of various forms. Obviously there are trade-offs in terms of complexity and data size (cost) and improvements to phase detection and prediction. In order to select appropriate properties and pattern building strategies we implemented a variety of heuristics and evaluated them quantitatively using the metrics developed in Section 4. Here we present our most successful and general approach; this design summarizes low-level behaviour using (short) bit-vectors that encode the overall pattern of variation. This is a "second order" approach, considering variation in hardware event counts rather than absolute counts themselves as the basic unit. Translating hardware event data to bit-vector patterns involves first coarsening the (variation in) data into discrete "levels," and then building a corresponding bit-vector "shape" representation.

- "Levels": A basic discretization is applied to (variations in) event density data to coarsen the data and help identify changes that indicate significant shifts in behavior. We compute the density of events over time for each sample. By comparing the density of the current sample with that of the previous sample, we obtain a variation V. The variation V is discretized to to a corresponding level, P_V. In our experiments we use 4 discrete levels.
- Pattern "shapes" are then determined by observing the direction of changes, positive or negative, between consecutive samples. Complexity in shape construction is mainly driven by determining when a pattern begins or ends.

  Each shape construction is represented by a pair $(P_V, \overline{v})$, where P_V is a level associated with the beginning of the shape, and $\overline{v}$ is a bit-vector encoding the sign (positive, negative) of successive changes in event density. Given data with level P_V, if there is no shape under construction a new construction begins with an empty vector: $(P_V, [\,])$. Otherwise, there must be a shape under construction $(Q_W, \overline{v})$. If $Q_W = P_V$, or we have seen $Q_W > P_V$ less than n times in a row, then shape creation continues based on the current shape construction $(Q_W, \overline{v})$: a bit indicating whether $V > 0$ or not is added to the end of $\overline{v}$.

  The following conditions terminate a shape construction:

  1. If we find $Q_W < P_V$ we consider the current shape building complete and begin construction of $(P_V, [\,])$. Increases in variation of event density are indicative of a significant change in program behavior.
  2. If we find $Q_W > P_V$, n times in a row the current shape has "died out." In this case we also consider the current shape building complete. In our experiments we use $n = 2$.
  3. If in $(Q_W, \overline{v})$ we find $|\overline{v}|$ has reached a predefined maximum length we also report the current construction as complete. In our experiments we use a maximum of 10 bits.

A rough overview of the pattern creation algorithm is shown in Figure 2. After obtained hardware data D, we compute the variation V between D and the same data (D_{last}) for the previous interval. V is then mapped from a real value to an integer value $P_V \in \{0, \ldots, n\}$, representing the "level" of V. As shown

in the formal description of this algorithm, we use Q_W to represent the level of the pattern currently under construction. Initially the value of Q_W is set to -1 to indicate no pattern is under construction. If $P_V > Q_W$ then we are facing a larger, and hence more important variation than the one that began the current pattern construction. The current pattern is thus terminated and and a new pattern construction associated with level P_V is begun. The value of P_V is assigned to Q_W and the shape code vector (denoted as *ShapeCode* in Figure 2) is blanked. Otherwise ($P_V \leq Q_W$) and the current pattern building continues.

The actual pattern encoding is based on the relation between P_V, Q_W and the sign of V. Two bits will be appended to the current *ShapeCode* each time a pattern grows: 01 means a positive variation at level Q_W, 10 represents a negative variation at level Q_W, and 00 means either a positive or negative variation at a level below Q_W. Binary 1s in our scheme thus indicate points of significant change. Construction continues until one of the pattern terminate conditions is met, at which point we report the pattern to the pattern analysis model. A concrete example of the creation of a pattern is shown in Figure 3.

Of course choice of primary data is also important; the above strategy can be applied to many different hardware events. In our actual system we make use of the instruction cache miss density as a good indicator of code activity. Use of other events and combined events is part of future work.

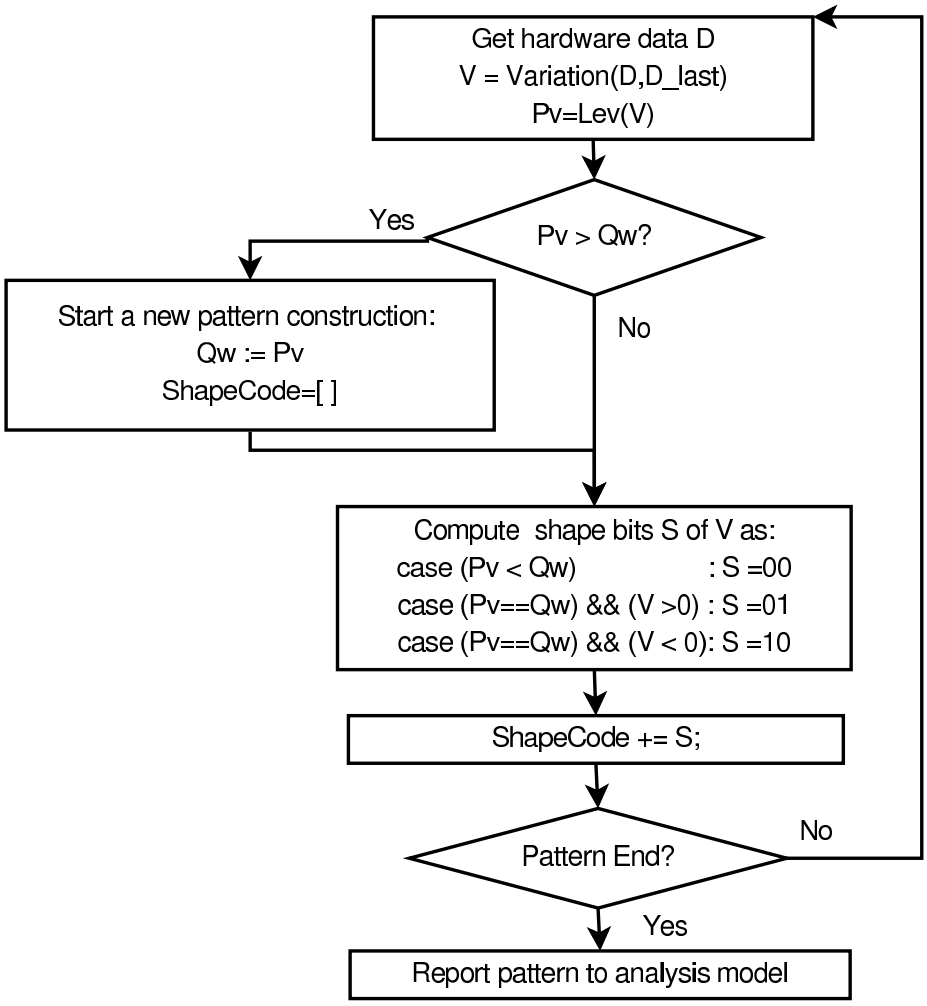

Figure 2: A flow chart for pattern creation.

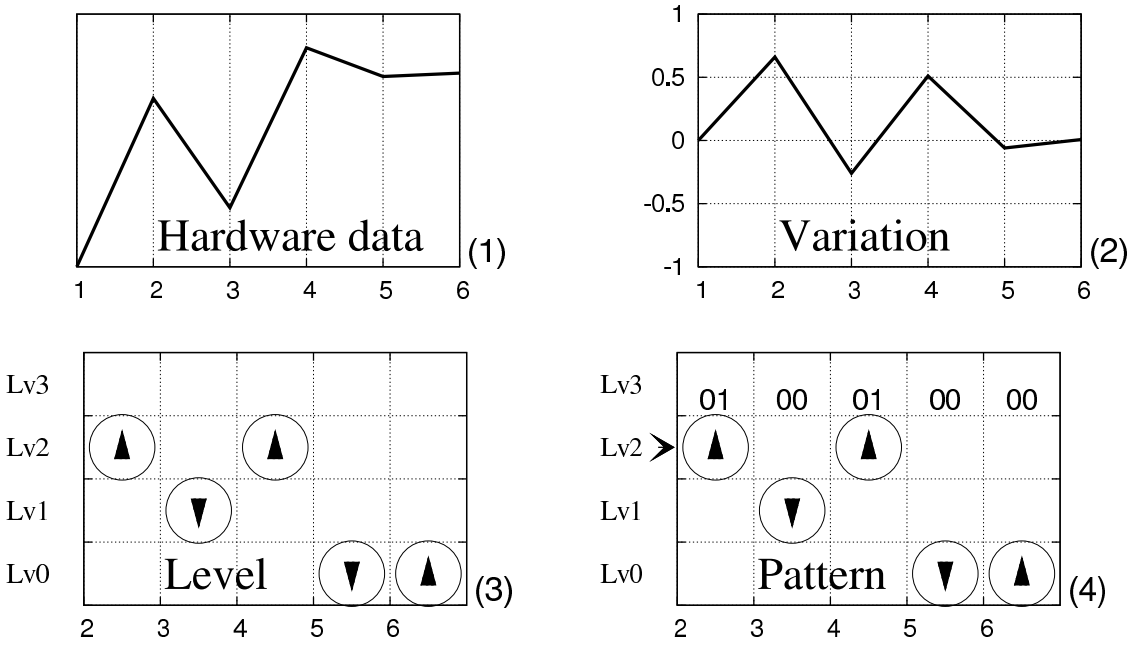

Figure 3: Pattern creation example. (1) Acquire the raw hardware data. (2) Calculate the variation between consecutive points. (3) Coarsen the variation into different levels; the triangles inside each circle show the direction (negative/positive) of variation. (4) The final pattern creation results; the arrow on the y-axis indicates that we obtain a level 2 pattern; the number above each circle shows the 2-bit code for each variation. The four trailing zeros are omitted (the pattern has died out), and the final pattern code is 010001.

3.2 Pattern analysis and prediction

Pattern analysis and prediction consumes patterns generated by the pattern creation module. Here we further examine the patterns to discover repetitive phases and generate predictions of future program behaviour. All created patterns are stored into a *pattern database*. The recurrent pattern detection and prediction are based on the information in the pattern database and the incoming pattern.

The recurrent detection is straightforward: if we find a newly created pattern that shares the same pattern code as a pattern stored in the pattern database we declare it to have recurred. An actual repetitive phase, however, is not declared unless the current pattern also matches the prediction results.

The prediction strategy we use is a variant of fixed-length, local/global mixed history, table-based prediction. Unlike more direct table-based methods our predictions include an attached "confidence" value; this allows us to track multiple prediction candidates and select the most likely.

Figure 4 gives an overview of our prediction scheme. For each pattern, we keep the three most popular repetition "distances" from a former occurrence to a later one—the use of three candidates is based on experimentally balancing predictor performance and accuracy. Prediction updates are performed by heuristically evaluating these distances for a given incoming pattern to find the most likely, variable-length pattern repetition. Our *tri-distance selection algorithm* updates the likely choices for an incoming pattern p by tracking three repetitions D_i, $i \in \{0, 1, 2\}$:

- For each D_i we keep a repetition length L_i, measured by subtracting time stamps of occurrences,

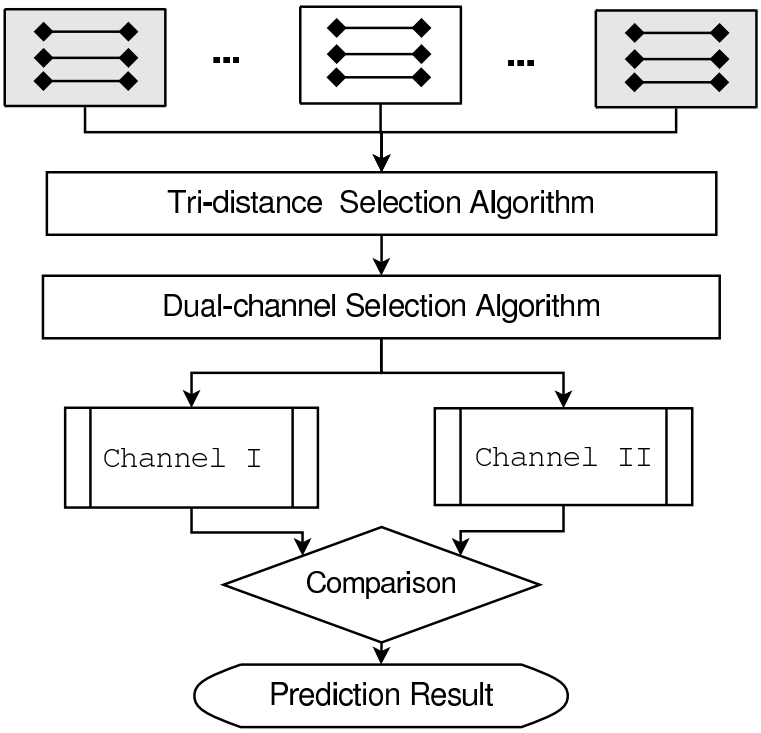

Figure 4: Overview of the prediction mechanism.

and a "hotness" value H_i.

- The difference T_i between the current pattern occurrence p and the ending point of each of D_i is calculated.
- If the difference between T_i and L_i is smaller than a threshold T, the hotness H_i is increased. Otherwise, H_i is decreased.
- If the difference between T_i and L_i is larger than T for all three D_i, we replace the D_j associated with the lowest hotness with a new D_j. The length, L_j is based on the distance to the closest of the current set of D_i, and hotness, H_j, is initialized to a constant value representing a low but positive hotness in order to give the new pattern a chance to become established.
- We use the D_i with the greatest hotness as the prediction result; H_i further functions as a confidence value for this prediction.

With the current prediction updated we then make a final prediction from the global set of pattern updates. In this case we use two global prediction "channels" to limit the cost of choosing among all possible patterns. Our *dual-channel selection algorithm* is similar to the tri-distance selection algorithm: if the current prediction matches one or both of the prediction channels the channel hotness is increased by the prediction confidence, and if it matches neither then the coldest channel is replaced. The hottest channel then determines the global prediction result.

3.3 Controlling the runtime profiling

For our application example we use the repetitive phase detection and prediction results to control the normal runtime profiling mechanism of Jikes RVM used to guide its adaptive optimization system. When there is no recurrent pattern the runtime measurement component takes profiles as usual. When a recurrent pattern is detected we compare it with the previous prediction. If it changes the prediction result we still keep collecting profiles, but also save them into an extra, variable-length local buffer. If the predicted pattern is the same as the last prediction we stop the profiling and instead "replay" the samples in the local buffer. Real program behaviour can of course drift from predicted behaviour over time, and so to ensure profiling accuracy we have a count-down, rechecking scheme to re-enable the profiling periodically irrespective of prediction.

4 Evaluation Metrics

Evaluation of phase analysis and prediction is typically done through either visual inspection or through impact on some external optimization. Here we provide two novel sets of metrics. The first evaluates the quality of repetition detection, and the second measures the accuracy and workload reduction of our profiling consumer application.

4.1 Quality of repetition detection metrics

Although the repetitive behaviour of programs has been known for a long time there is a general lack of a formal ways to quantitively evaluate it. This is especially true of long term, variable-length program periodicity as we investigate here. We thus define two metrics, *Confidence* and *Possible Miss Rate (PMR)*. *Confidence* gives a measure of the similarity between repetitive periods identified by our algorithm, while *PMR* measures the amount of execution which could have been identified as repetitive but which was not done so by the phase detection algorithm. Both Confidence and PMR are based on the same pair of fundamental metrics measuring the similarity between execution segments that may be allocated to the same repetitive group, such as the instances (occurrences) of the same pattern.

Suppose we have a pattern P which has N instances. The group of all the instances of P can be represented as an ordered set, $G(P) = \{P_i | i = 1, 2, ..., N\}$ Two basic metrics are then used to quantify the *similarity* and *regularity* of a set $G(P)$.

- *Similarity:* We calculate the *Pearson correlation* between each pair in $G(P)$ and use the average of the results to represent the similarity of a group. We denote this value as $C_{G(P)}$.
- *Regularity:* The time difference between start times for each pair of adjacent P_i provides a basic "distance" measure between pattern instances. The extent to which pattern instances are well clustered shows regularity; we measure it using a *k-means* clustering algorithm [20] applied to the set of all distance pairs. For each cluster we obtain the absolute value of the difference between each item and the centroid of the cluster. The sum of all these difference becomes a measure of the regularity of the pattern group $G(P)$, and we denote this value as $D_{G(P)}$.

Combining the above calculations, we provide an overall evaluation of $G(P)$ as:

$$E_{G(P)} = C_{G(P)} * D^{-1}_{G(P)}$$

Given different repetition detections for the same pattern P a higher $E_{G(P)}$ heuristically indicates better results.

Our actual metrics can now be defined in terms of the above calculations.

- *Confidence:* For each $G(P)$, we generate a set $\hat{G}(P)^j$ by removing the j^{th} pattern instance of $G(P)$. If $\hat{G}(P)^j$ has a better quality (E) than $G(P)$, then we have less confidence on the j^{th} pattern instance being a member of the group, and thus reduced confidence in the grouping itself. Otherwise, the j^{th} instance makes the whole group better and improves confidence.

  We thus give a confidence score *Conf(P_j)* of the j^{th} item of $G(P)$ as:

$$Conf(P_j) = \begin{cases} 1.0 & E_{G(P)} > E_{\hat{G}(P)^j} \\ \frac{E_{G(P)}}{E_{\hat{G}(P)^j}} & \text{Otherwise} \end{cases}$$

  Confidence in the detection results of pattern P, denoted as $Conf(P)$, is then the sum of *Conf(P_j)* for all j.

  Our final *Confidence* in a complete detection result on all m patterns $P^1, P^2, ..., P^m$ appearing in the result is the sum of confidence in each pattern weighted by the "size" of the pattern:

$$\textit{Confidence} = \frac{\sum_{i=1}^{m} \textit{Conf}(P^m)}{\sum_{i=1}^{m} |G(P^m)|}$$

  Confidence basically indicates the degree to which the pattern detection results represent at least a local maximum. High confidence indicates patterns are well-categorized, while low confidence suggests some execution segments may be misclassified.

- *Possible miss rate (PMR):* The *PMR* evaluates how much of the execution was potentially misidentified as non-repetitive. We define it as follows:

$$\textit{PMR} = \frac{\textit{Number of PMPI}}{\textit{Number of PMPI} + \textit{Number of DPI}} \quad (1)$$

  In formula 1 above, *PMPI* stands for "Possible Missed Pattern Instances" and *DPI* represents "Detected Pattern Instances". Somewhat dual to *Confidence,* the fundamental idea of *PMR* is to add an execution segment as an instance of a pattern and check whether this new grouping is better or worse.

  Given a pattern detection result $G(P)$, we treat all the execution segments that are not covered by $G(P)$ as potential elements of *PMPI.* We then insert each such execution segment s into $G(P)$ and build a new group $\check{G}(P)^s$. Segment s is then included as a member of *PMPI* if $E_{G(P)} < E_{\check{G}(P)^s}$.

4.2 Profiling metrics

The two metrics given in Section 4.1 evaluate our repetitive phase detection in terms of distance from an abstract ideal. Here we describe how we evaluate the success of phase prediction when applied to a concrete optimization.

Our example application is an improvement to the runtime profiling in Jikes RVM used to support its adaptive compiler [2]. This profiler samples execution periodically in order to identify "hot methods" and make (re)compilation decisions; sampling rates heuristically trade off accuracy for profiling cost. We provide two metrics for evaluating the impact of phase prediction on profiling:

- *Profiling rate:* Profiling rate P_r is defined as:

$$P_r = \frac{\textit{Number of Actual Profiling Points}}{\textit{Number of All Possible Profiling Points}} * 100\%$$

An unmodified version of the runtime profiling mechanism has a P_r of 100%. Based on phase predictions, we disable some profiling points; a lower value of P_r indicates a reduction in the profiling workload.

- *Coverage score (Cov):* The Jikes RVM profiler makes use of the relative number of probe results in each method. Our predicted results should thus produce the same intended effect.

A method profiling result R on methods $M_i, i \in \{1, \ldots, m\}$ can be represented as:

$$R = \{< M_i, \textit{Per}_i^R >\}$$

where $\textit{Per}_i^N$ is the percentage ratio of samples in method M_i to the total number of program samples. Given a canonical sample result $N = \{< M_i, \textit{Per}_i^N >\}$. The *Cov* of R is calculated as:

$$\textit{Cov}(R) = \sum_{i=1}^{m} \textit{Min}(\textit{Per}_i^R, \textit{Per}_i^N)$$

To compare the accuracy of phase based profiling to the original profiling results we obtain a canonical N by averaging multiple standard executions of the original profiling mechanism. In practice N is reasonably stable. The *Cov* for a phase based profiling run compared with the average *Cov* of each of our standard runs provides an *accuracy score* that indicates how much a given phase based profile varies from typical runs.

5 Experimental Analysis

Here we make use of the metrics developed in the previous section to experimentally evaluate our technique. Following our experimental setting we first present our quality results, followed by our profiling workload and accuracy measurements.

5.1 Setting and Benchmarks

Our implementation is based on JikesRVM 2.3.6 with an adaptive JIT compiler; results were measured on an Athlon 1.4GHz workstation with 1GB memory (Debian Linux, 2.6.9 kernel). We report phase detection results derived from L1 instruction cache miss events. Benchmarks include the industry standard SPECJVM98 suite [26], and two larger examples, SOOT and PSEUDOJBB. SOOT is a Java optimization framework which takes Java class files as input and applies optimizations to the bytecode; in our experiments, we run SOOT on the class files for JAVAC in SPECJVM98 with options "–app -O". PSEUDOJBB is a variant of SPECJBB2000 [27] which executes a fixed number of transactions in multiple warehouses. Our experiments run one to eight warehouses, 100 000 transactions in each warehouse. For SPECJVM98 we use the recommended (large) input size "-s 100". For quality analysis we built a canonical sample profile from 15 typical runs, while the phase driven profiling results are the average of 5 runs. The threshold T for tri-distance selection is set to 10%.

5.2 Results

Columns 2 through 6 in Table 1 show the metric results, calculated using an offline analysis on trace files from our online implementation. The five data columns are the number of different patterns, the number of occurrences of all patterns, *Confidence* results, *PMR* results and *PMR* results on the most important (major level) patterns.

Bench.	Number of Patterns	Number of Occur.	Conf.	PMR (%)	PMR Major(%)	Profiling Rate (%)	Accuracy Score (%)	
							Phase Driven	Simple 50%
compress	32	158	0.94	60.78	2.78	52.2	91.72	91.71
db	29	451	0.95	35.94	1.25	37.5	85.61	89.54
jack	29	352	0.94	22.65	0.05	46.0	95.55	68.56
javac	23	214	0.93	32.42	6.58	54.8	99.32	76.87
jess	25	182	0.88	48.71	5.88	47.3	91.92	79.12
mpegaudio	28	111	0.91	68.71	13.49	49.7	92.47	83.76
mtrt	27	78	0.85	27.58	0.10	77.7	97.15	83.00
raytrace	18	69	0.85	16.17	4.44	97.9	99.97	83.82
soot	49	11106	0.99	28.45	0.03	27.2	93.83	61.43
PseudoJbb	35	7093	0.98	37.80	0.01	30.0	94.71	64.84
Average	—	—	0.92	37.98	3.46	51.02	94.31	78.26

Table 1: Pattern detection evaluation and profiling workload reduction results(*Occur.* for Occurrence, *Conf.* for Confidence)

On average we have a 92% *Confidence* that the segments identified by our algorithm are actual repetitive portions. Unfortunately we also have a comparatively high average *PMR,* 38%. This means we potentially miss over a third of repetitive segments in the execution. Deeper investigation shows that most of the missed segments are likely instances of patterns at the lowest levels (0 and 1). As described in Section 3, pattern creation at lower levels will be interrupted when a higher level change is encountered. It is not therefore surprising that many possible repetitions of lower level patterns are ignored by our algorithm; larger, more significant changes are expected to be more important for capturing the important repetitive behavior of a program, and our algorithm weights such patterns higher. In Table 1, the "PMR Major" column gives the PMR value for only the upper range of variance (levels 2 and 3). For these signals the data shows that we only miss on average about 3.5% of possible repetitive periods.

Profiling workload reduction and accuracy results are shown in the last three columns of Table 1. On average we reduce the profiling workload by about a half, although results vary significantly by benchmark. Profiling accuracy, however, is uniformly very high; on average we achieve a 94.3% accuracy, profiling at 51% of possible profiling points. For comparison purposes we show the accuracy score for a simple profiling reduction strategy that just omits every other probe, also a factor of 2 workload reduction. On benchmarks with small hot method sets, such as COMPRESS and DB, profiling results are not sensitive to profiling rate. On more complicated benchmarks, such as JACK, SOOT and PSEUDOJBB our technique is significantly more accurate, usually with less than a 50% profiling rate. These results are also shown in Figure 5.

6 Conclusions & Future Work

Understanding repetitive program behaviour and exploring phases in program execution is interesting for researchers in runtime techniques, program simulation and static/runtime compilation. For most purposes a

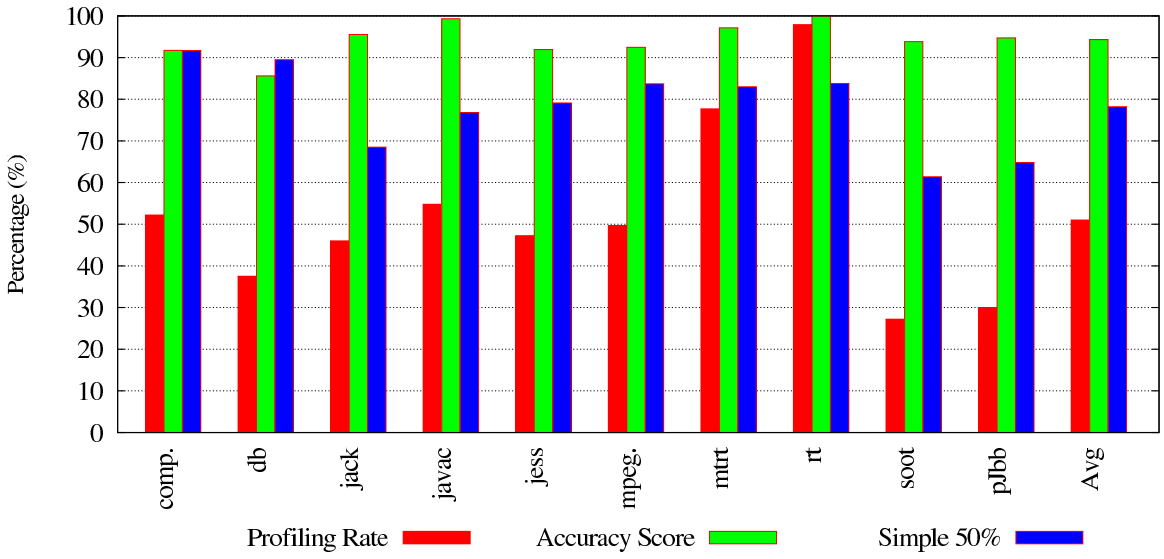

Figure 5: Profiling workload reduction and accuracy results.

high quality phase detection is important, but this is difficult or expensive to acquire from pure software-level designs. Comparable, quantitative ways of evaluating phase prediction have also not been well explored.

In this paper, we present a new lightweight technique for determining and predicting repetitive phases in program execution. Our approach builds on hardware event data, ensuring a close relation to actual runtime performance. We evaluate the performance of our system using a novel set of well-defined metrics that indicate different, important aspects of quality in detected phases. To show the utility of this information, we present an optimized, phase-driven runtime profiling mechanism as a sample application; with phase prediction significant reductions in profiling workload are possible while still ensuring high accuracy.

Of course repetitive phase information can be used in a wide range of areas other than selective profiling. Our future work involves application of phase information to directly controlling runtime recompilation decisions, optimization strategy choices, adaptive system reconfiguration, and selection of garbage collection points.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, Oct. 1999.

[2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 111–129, New York, NY, USA, 2002. ACM Press.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[4] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *MICRO 33:the 33rd Annual Intl. Sym. on Microarchitecture*, pages 245–257, Dec. 2000.

[5] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W. Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 233–244, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] S. Brown, J. Dongarra, N. Garner, K. London, and P. Mucci. PAPI. `http://icl.cs.utk.edu/papi`.

[7] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):617–637, 2005.

[8] A. Dhodapkar and J. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines, 2002.

[9] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244. IEEE Computer Society, 2002.

[10] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220. IEEE Computer Society, Sep. 2003.

[11] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, Oct. 2004.

[12] D. Gu and C. Verbrugge. A survey of phase analysis: Techniques, evaluation and applications. Technical Report SABLE-TR-2006-1, Sable Research Group, McGill University, March 2006.

[13] D. Gu, C. Verbrugge, and E. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, New York, NY, USA, June 2006. ACM Press.

[14] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase shift detection: A problem classification. Technical Report IBM Research Report RC-22887, IBM T. J. Watson, August 2003.

[15] IBM. Pmapi. `http://www.alphaworks.ibm.com/tech/pmapi`.

[16] D. A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 107–116, New York, NY, USA, 2005. ACM Press.

[17] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[18] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals to find hierarchical phase behavior. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, March 2005.

[19] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, page 220. IEEE Computer Society, March 2005.

[20] J. McQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and N. Neyman, editors, *the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297, 1967.

[21] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2)*, pages 40–47, Oct. 2004.

[22] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 189–198, Oct. 2004.

[23] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. *SIGPLAN Not.*, 39(11):165–176, 2004.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[25] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, 2003.

[26] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks. `http://www.spec.org/osg/jvm98`.

[27] Standard Performance Evaluation Corporation. SPECjbb2000. `http://www.spec.org/osg/jbb2000`, 2000.

[28] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195, New York, NY, USA, 2001. ACM Press.

[29] Sun Microsystems, Inc. The Java Virtual Machine Tools Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[30] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM'04:Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.